On Matrices, Automata, and Double Counting in Constraint Programming

Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, Justin Pearson

Constraints manuscript No.
(will be inserted by the editor)

On Matrices, Automata, and Double Counting in Constraint Programming

Nicolas Beldiceanu · Mats Carlsson · Pierre Flener · Justin Pearson

Received: 30 August 2011 / Revised: 28 March 2012 / Accepted: 21 November 2012

Abstract Matrix models are ubiquitous for constraint problems. Many such problems have a matrix of variables $\mathcal{M}$, with the same constraint C defined by a finite-state automaton $\mathcal{A}$ on each row of $\mathcal{M}$ and a global cardinality constraint gcc on each column of $\mathcal{M}$. We give two methods for deriving, by double counting, necessary conditions on the cardinality variables of the gcc constraints from the automaton $\mathcal{A}$. The first method yields linear necessary conditions and simple arithmetic constraints. The second method introduces the *cardinality automaton*, which abstracts the overall behaviour of all the row automata and can be encoded by a set of linear constraints. We also provide a domain consistency filtering algorithm for the conjunction of lexicographic ordering constraints between adjacent rows of $\mathcal{M}$ and (possibly different) automaton constraints on the rows. We evaluate the impact of our methods in terms of runtime and search effort on a large set of nurse rostering problem instances.

Keywords Double counting · necessary (implied) constraint · matrix model · automaton constraint · nurse scheduling

This paper extends a prior version published as [2].

N. Beldiceanu
TASC team (CNRS/INRIA), Mines de Nantes, 44307 Nantes, France
E-mail: Nicolas.Beldiceanu@mines-nantes.fr

M. Carlsson
SICS, P.O. Box 1263, 164 29 Kista, Sweden
E-mail: Mats.Carlsson@sics.se

P. Flener · J. Pearson
Uppsala University, Department of Information Technology, Box 337, 751 05 Sweden
E-mail: Pierre.Flener@it.uu.se, E-mail: Justin.Pearson@it.uu.se

1 Introduction

Matrix models are ubiquitous for constraint problems. Despite this fact, only a few constraints consider a matrix and some of its constraints as a whole: the $allperm$ [13] and $lex2$ [10] constraints were introduced for breaking symmetries in a matrix, while the $colored\_matrix$ constraint [20] was introduced for handling a conjunction of gcc constraints[1] on the rows and columns of a matrix. We focus on another recurring pattern, especially in the context of personnel rostering, which can be described in the following way.

Given three positive integers R, K, and V, we have an $R \times K$ matrix $\mathcal{M}$ of decision variables that take their values within the finite set of values $\{0, 1, \ldots, V-1\}$, as well as a $V \times K$ matrix $\mathcal{M}^{\#}$ of cardinality variables that take their values within the finite set of values $\{0, 1, \ldots, R\}$. Each row r (with $0 \leq r < R$) of $\mathcal{M}$ is subject to a constraint defined by an automaton[2] $\mathcal{A}$ and, depending on the search procedure, we may break symmetries by a lexicographic ordering between adjacent rows [7, 11, 12]. For simplicity (except in Section 5), we assume that each row is subject to the same constraint. Each column k (with $0 \leq k < K$) of $\mathcal{M}$ is subject to a gcc constraint that restricts the number of occurrences of the values according to column k of $\mathcal{M}^{\#}$: let $\#_k^v$ denote the number of occurrences of value v (with $0 \leq v < V$) in column k of $\mathcal{M}$, that is, the cardinality variable in row v and column k of $\mathcal{M}^{\#}$. We call this pattern the *matrix-of-automaton-and-gcc* pattern. We also introduce an $R \times V$ matrix $\mathcal{M}'^{\#}$ of cardinality variables that take their values within the finite set of values $\{0, 1, \ldots, K\}$. Each row r (with $0 \leq r < R$) of $\mathcal{M}$ is also subject to a gcc constraint, derived from the finite-state automaton, that restricts the number of occurrences of the values according to row r of $\mathcal{M}'^{\#}$: let $\#_v'^r$ denote the number of occurrences of value v (with $0 \leq v < V$) in row r of $\mathcal{M}$, that is, the cardinality variable in column v and row r of $\mathcal{M}'^{\#}$. In the context of personnel rostering, a possible interpretation of this pattern is:

- R, K, and V respectively correspond to the number of persons, days, and types of work (e.g., *morning shift*, *afternoon shift*, *night shift*, or *day off*) we consider.
- Each row r of $\mathcal{M}$ corresponds to the work of person r over K consecutive days.
- Each column k of $\mathcal{M}$ corresponds to the work by the R persons on day k.
- The automaton $\mathcal{A}$ on the rows of $\mathcal{M}$ encodes the rules of a valid schedule for a person; it can be the product of several automata defining different rules.
- The gcc constraint on column k represents the demand of services for day k. In this context, the cardinality associated with a given service can either be fixed or be specified to belong to a given range.

A typical problem with this kind of pattern is the lack of interaction between the row and column constraints. This is especially problematic when, on the one hand,

[1] Given a set of decision variables $vars$ and a set of value-variable pairs val_occ, the $gcc(vars, val\_occ)$ constraint enforces for each value-variable pair $val : occ$ of val_occ that val occur exactly occ times within $vars$. Moreover, it imposes that all variables of $vars$ be assigned a value from val_occ.

[2] The $automaton(X, \mathcal{A})$ constraint [3] requires the sequence X of decision variables to take values that, seen as a string, are accepted by the finite-state automaton $\mathcal{A}$, which is possibly augmented with counters. In the absence of counters, this is equivalent to the $regular(X, \mathcal{A})$ constraint [19].

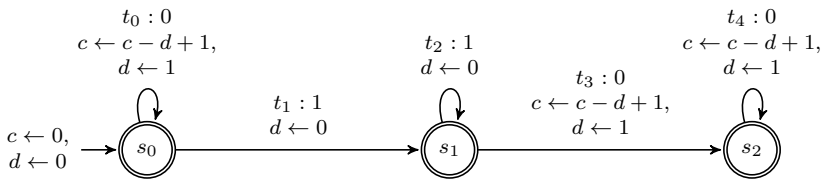

Figure 1 Automaton $\mathcal{C}$ associated with the $global\_contiguity$ constraint, with initial state s_0, accepting states s_0, s_1, s_2, and transitions t_0, t_1, t_2, t_3, t_4 labelled by values 0 or 1. The missing transition for value 1 from state s_2 is assumed to go to a dead state. The automaton has been annotated with counters [3]: the final value of counter c is the number of stretches of value 0, whereas d is an auxiliary counter.

the row constraint is a sliding constraint expressing a distribution rule on the work, and, on the other hand, the demand profile (expressed with the gcc constraints) varies drastically from one day to the next (e.g., during weekends and holidays in the context of personnel rostering). This issue is usually addressed by experienced constraint programmers by manually adding necessary conditions (implied constraints), which are typically based on some simple counting conditions depending on some specificity of the row constraints. Let us first introduce a toy example to illustrate this phenomenon.

We show that implied constraints can be derived by using the combinatorial technique of *double counting* (see for example [15]). We use the two-dimensional structure of the matrix, counting along the rows and the columns. Some feature is considered, such as the number of appearances of a word or stretch, and the occurrences of that feature are counted for the rows and columns separately. When the counting is exact, these two values will coincide. In order to derive useful constraints that will propagate, we derive lower and upper bounds on the given feature occurring when counted column-wise. These are then combined into inequalities saying that the sum of these column-based lower bounds is *at most* the sum of given row-based upper bounds, or that the sum of these column-based upper bounds is *at least* the sum of given row-based lower bounds.

Example 1 Take a 3×7 matrix $\mathcal{M}$ of 0/1 variables (i.e., $R = 3$, $K = 7$, $V = 2$), where on each row we have a $global\_contiguity$ constraint (all the occurrences of value 1 are contiguous) for which Figure 1 depicts a corresponding automaton $\mathcal{C}$ (the reader can ignore the assignments to counters c and d at this moment). In addition, $\mathcal{M}^{\#}$ defines the following gcc constraints on the columns of $\mathcal{M}$:

- Columns 0, 2, 4, and 6 of $\mathcal{M}$ must each contain two 0s and a single 1.
- Columns 1, 3, and 5 of $\mathcal{M}$ must each contain two 1s and a single 0.

A simple double counting argument proves that there is no solution to this problem. Indeed, consider the sequence of numbers of occurrences of 1s on the seven columns of $\mathcal{M}$, that is $1, 2, 1, 2, 1, 2, 1$. Each time there is an increase of the number of 1s between two adjacent columns, a new stretch of consecutive 1s starts on at least one row in the second of these columns of the matrix. From this observation we can deduce that we have at least four stretches of consecutive 1s, namely one stretch starts at the first column (since implicitly before the first column we have zero occurrences

of value 1) and three stretches start at the columns containing two 1s. But since we have a $global\_contiguity$ constraint on each row of the matrix and since the matrix only has three rows, there is a contradiction. □

After giving a first basic use of double counting (Section 2), the contributions of this paper include:

- Methods for deriving necessary conditions on the cardinality variables of the gcc constraints from (combinations of) string properties that hold for an automaton $\mathcal{A}$ (Sections 3.1 to 3.5), including when the gcc constraints on the columns are replaced by summation constraints (Section 3.6).
- A method for annotating an automaton $\mathcal{A}$ with counter variables extracting string properties from $\mathcal{A}$ (Section 3.7), and a heuristic for selecting relevant string properties (Section 3.8).
- Another method for deriving necessary conditions on the cardinality variables, called the *cardinality automaton*, which simulates the overall behaviour of all the row automata (Section 4).
- A method for achieving domain consistency on a chain of lexicographic ordering constraints augmented with an arbitrary *automaton* constraint on every element of the chain (Section 5).
- An evaluation of the impact of our methods in terms of runtime and search effort on a large set of nurse rostering problem instances (Section 6).

2 Basic Double Counting

We now give a first basic use of double counting on matrix $\mathcal{M}$. As sketched in the introduction, we use for each column k (with $0 \leq k < K$) and each row r (with $0 \leq r < R$) of $\mathcal{M}$ a gcc constraint for linking the variables of a column of $\mathcal{M}$ and the variables of a row of $\mathcal{M}$ with the occurrence variables of the corresponding column of $\mathcal{M}^{\#}$ and the occurrence variables of the corresponding row of $\mathcal{M}'^{\#}$. Let us introduce for each value in the finite set $\{0, 1, \ldots, V-1\}$ a counting variable C_v (with $0 \leq v < V$) that denotes how many entries of matrix $\mathcal{M}$ are assigned value v. We have:

$$\forall v \in [0, V-1] : C_v = \sum_{k=0}^{K-1} \#_k^v \tag{1}$$

$$\forall v \in [0, V-1] : C_v = \sum_{r=0}^{R-1} \#_v'^r \tag{2}$$

$$\sum_{v=0}^{V-1} C_v = R \cdot K \tag{3}$$

Equation (3) may allow us to tighten the bounds of the counting variables C_v (with $0 \leq v < V$), especially when some bounds of the counting variables come from propagating Equation (1), while others come from propagating Equation (2).

3 Deriving Necessary Conditions from String Properties

We now develop a first method for deriving necessary conditions for the *matrix-of-automata-and-gcc* pattern. The key idea is to approximate the set of solutions to the row constraint C by string properties such as the following:

- Bounds on the number of letters, words, prefixes, or suffixes (see Section 3.1).
- Bounds on the number of stretches of a given value (see Section 3.2).
- Bounds on the lengths of stretches of a given value (see Section 3.3).
- The combination of forbidden prefixes or suffixes with bounds on the number of stretches of a given value (see Section 3.4).
- Value precedence relations between specific pairs of values in any solution to C (see Section 3.5).

We first develop a set of formulae expressed in terms of simple arithmetic constraints for such string properties. Each formula gives a necessary condition for the *matrix-of-automata-and-gcc* pattern provided that the set of solutions to the row constraint satisfies a given string property. We then show how to adapt these results when the gcc constraints on the columns are replaced by summation constraints (see Section 3.6). The hurried reader can jump at any time to Section 3.7, but should note that many of the string properties we consider occur naturally in the context of timetabling problems, such as the one of Section 6.

We also show how to extract automatically such string properties from an automaton (see Section 3.7 and outline a heuristic for selecting relevant string properties (see Section 3.8). String properties can be seen as a communication channel for enhancing the propagation between row and column constraints.

A key advantage of the overall approach described in this section is its incremental nature, which depends on a set of string properties and formulae that can be refined and enriched over time in order to get strong necessary conditions.

3.1 Constraining the Number of Occurrences of Words, Prefixes, and Suffixes

A *word* is a fixed sequence of values, seen as letters. Suppose we have the following bounds for each row r on how many times a given word occurs (possibly in overlapping fashion) in that row, denoted by $W_r(w)$, all numbering starting from zero:

- $LW_r(w)$ is the minimum number of times that the word w occurs in row r (i.e., $W_r(w) \geq LW_r(w)$).
- $UW_r(w)$ is the maximum number of times that the word w occurs in row r (i.e., $W_r(w) \leq UW_r(w)$).

Note that letters are just singleton words. It is not unusual for $LW_r(w)$ (or $UW_r(w)$) to be equal for all rows r for a given word w. From this information, we now infer by double counting two necessary conditions for each such word.

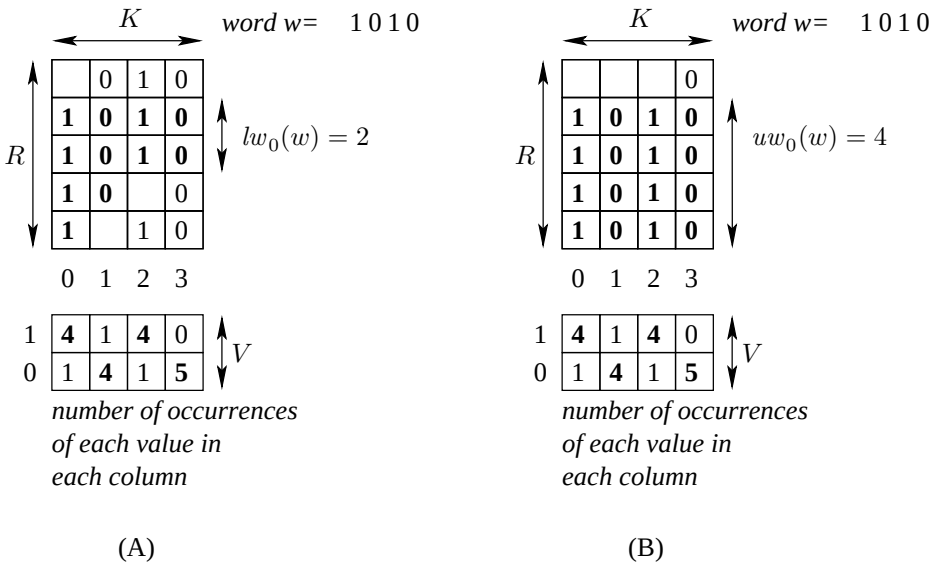

Figure 2 Lower and upper bounds on the number of words starting at a given column. Boldface in the $R \times K$ matrices corresponds to partial instances of the word $w = 1010$ for which we try to minimise (A) or maximise (B) the number of occurrences. Boldface in the $V \times K$ matrices corresponds to letters of the word $w = 1010$.

3.1.1 Necessary Conditions

Let $|w|$ denote the length of word w, and let w_j denote the letter at position j in word w. The following bounds:

$$lw_k(w) = \max\left(\left(\sum_{j=0}^{|w|-1} \#_{k+j}^{w_j}\right) - (|w|-1)\cdot R, 0\right) \tag{4}$$

$$uw_k(w) = \min\left\{\#_{k+j}^{w_j} \mid 0 \leq j \leq |w|-1\right\} \tag{5}$$

correspond respectively to the minimum and maximum number of occurrences of word w that *start* at column $k \in [0, K-|w|]$; this number is denoted by $w_k(w)$ (i.e., $lw_k(w) \leq w_k(w) \leq uw_k(w)$). These bounds can be obtained as follows:

- Since the cardinality variables only denote the number of times a value occurs in each column and do not constrain *where* it occurs, the lower bound (4) is the worst-case intersection of all column value occurrences.
- A word cannot occur more often than its minimally occurring letter, hence bound (5).

Example 2 Parts (A) and (B) of Figure 2 respectively illustrate the lower and upper bounds expressed by equations (4) and (5) on the number of occurrences of word $w = 1010$ starting at column 0, provided that the numbers of 0 (respectively 1) in columns 0, 1, 2, 3 are respectively equal to 4, 1, 4, 0 (respectively 1, 4, 1, 5). □

Note that if some cardinality variable is not fixed, then equations (4) and (5) should be interpreted as arithmetic constraints. We get the following necessary condition:

$$\sum_{k=0}^{K-|w|} w_k(w) = \sum_{r=0}^{R-1} W_r(w) \tag{6}$$

Note also that while evaluating the maximum value of the left-hand side of equality (6), we may overestimate the maximum number of occurrences of word w since, for instance, if the first two letters of w are distinct, then the maximum number of occurrences of word w starting in two consecutive columns is also limited by R, and not just by $uw_k(w) + uw_{k+1}(w)$.

3.1.2 Generalisation: Replacing Each Letter by a Set of Letters

So far, all letters of the word w were fixed. We now assume that each letter of a word can be replaced by a finite nonempty set of possible letters. For this purpose, let w_j now denote the set of letters for position j of word w. Hence the bounds $lw_k(w)$ and $uw_k(w)$ are now defined by aggregation as follows:

$$lw_k(w) = \max\left(\left(\sum_{j=0}^{|w|-1} \sum_{c \in w_j} \#_{k+j}^{c}\right) - (|w|-1)\cdot R, 0\right) \tag{7}$$

$$uw_k(w) = \min\left\{\sum_{c \in w_j} \#_{k+j}^{c} \mid 0 \leq j \leq |w|-1\right\} \tag{8}$$

We get the same necessary conditions as before.[3] Note that (7) and (8) specialise respectively to (4) and (5) when all w_j are singleton sets.

3.1.3 Extension: Constraining Prefixes and Suffixes

We now consider constraints on a word occurring as a prefix (the first letter of the word is at the first position of the row) or suffix (the last letter of the word is at the last position of the row). Let $WP_r(w)$ (respectively $WS_r(w)$) denote the number of times word w is a prefix (respectively a suffix) of row r, and suppose we have the following bounds:

- $LWP_r(w)$ is the minimum number of times (0 or 1) word w is a prefix of row r.
- $UWP_r(w)$ is the maximum number of times (0 or 1) word w is a prefix of row r.
- $LWS_r(w)$ is the minimum number of times (0 or 1) word w is a suffix of row r.
- $UWS_r(w)$ is the maximum number of times (0 or 1) word w is a suffix of row r.

[3] When evaluating the number of occurrences $nocc_k^i$ of a set of letters associated to the potential value of the letter at position i of word w in column k, we should also use an $among(nocc_k^i, \langle \mathcal{M}[0,k], \mathcal{M}[1,k], \ldots, \mathcal{M}[R-1,k]\rangle, w_i)$ constraint in order to get a possibly sharper evaluation.

From these bounds, we get the following necessary conditions:

$$w_0(w) = \sum_{r=0}^{R-1} WP_r(w) \tag{9}$$

$$w_{K-|w|}(w) = \sum_{r=0}^{R-1} WS_r(w) \tag{10}$$

Note that these necessary conditions also hold when each letter of a constrained prefix or suffix is replaced by a set of letters.

3.2 Constraining the Number of Occurrences of Stretches

Given a sequence x of fixed variables and a value v, a *stretch* of value v is a maximum sequence of values in x that only consists of value v. Suppose now that we have bounds for each row r on how many times a stretch of a given value v can occur in that row, denoted by $S_r(v)$:

- $LS_r(v)$ is the minimum number of stretches of value v on row r (i.e., $S_r(v) \geq LS_r(v)$).
- $US_r(v)$ is the maximum number of stretches of value v on row r (i.e., $S_r(v) \leq US_r(v)$).

It is not unusual for $LS_r(v)$ (or $US_r(v)$) to be equal for all rows r for a given value v.

3.2.1 Necessary Conditions

The following bounds (under the convention that $\#_{-1}^v = 0$ for each value v)

$$ls_k^+(v) = \max(0, \#_k^v - \#_{k-1}^v) \tag{11}$$

$$us_k^+(v) = \#_k^v - \max(0, \#_{k-1}^v + \#_k^v - R) \tag{12}$$

correspond respectively to the minimum and maximum number of stretches of value v that *start* at column k, denoted by $s_k^+(v)$ (i.e., $ls_k^+(v) \leq s_k^+(v) \leq us_k^+(v)$). Again, *if some cardinality variable is not fixed, then the equations above should be interpreted as arithmetic constraints*. The intuitions behind these formulae are as follows:

- If the number of occurrences of value v in column k (i.e., $\#_k^v$) is strictly greater than the number of occurrences of value v in column $k-1$ (i.e., $\#_{k-1}^v$), then this means that at least $\#_k^v - \#_{k-1}^v$ new stretches of value v can start at column k.
- If the number of occurrences of value v in column k (i.e., $\#_k^v$) plus the number of occurrences of value v in column $k-1$ (i.e., $\#_{k-1}^v$) is strictly greater than the number of rows R, then the quantity $\#_{k-1}^v + \#_k^v - R$ represents the minimum number of stretches of value v that cover both column $k-1$ and column k. From this minimum intersection we get the maximum number of new stretches that can start at column k.

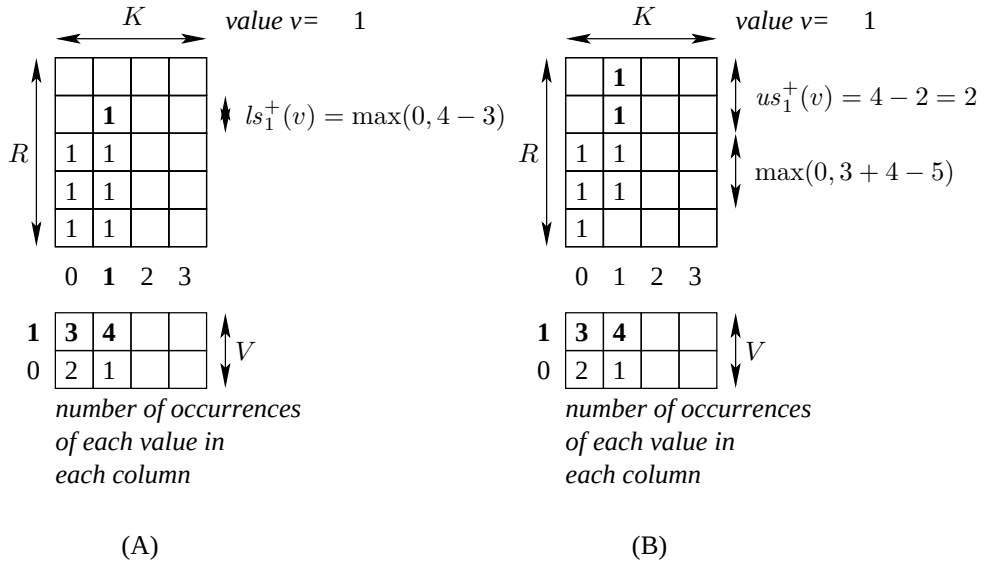

Figure 3 Lower and upper bounds on the number of stretches starting at a given column. Boldface in the $R \times K$ matrices corresponds to stretches of value 1 starting at column 1 that we are trying to minimise (A) or maximise (B). Boldface in the $V \times K$ matrices corresponds to the occurrence constraints on value $v = 1$.

Example 3 Parts (A) and (B) of Figure 3 respectively illustrate the lower and upper bounds expressed by equations (11) and (12) on the number of stretches of value 1 starting at column 1, provided that the number of occurrences of 0 (respectively 1) in columns 0 and 1 are equal to 2 and 1 (respectively 3 and 4). □

By aggregating these bounds for all the columns of the matrix, we get the following necessary condition using double counting:

$$\sum_{k=0}^{K-1} s_k^+(v) = \sum_{r=0}^{R-1} S_r(v) \tag{13}$$

Similarly, the following bounds (under the convention that $\#_K^v = 0$ for each value v)

$$ls_k^-(v) = \max(0, \#_k^v - \#_{k+1}^v) \tag{14}$$

$$us_k^-(v) = \#_k^v - \max(0, \#_{k+1}^v + \#_k^v - R) \tag{15}$$

correspond respectively to the minimum and maximum number of stretches of value v that *end* at column k, denoted by $s_k^-(v)$ (i.e., $ls_k^-(v) \leq s_k^-(v) \leq us_k^-(v)$). We get a similar necessary condition:

$$\sum_{k=0}^{K-1} s_k^-(v) = \sum_{r=0}^{R-1} S_r(v) \tag{16}$$

3.2.2 Generalisation: Replacing the Value by a Set of Values

So far, the value v of a stretch was fixed. We now assume that a stretch may consist of a finite nonempty set, denoted by $\hat{v}$, of possible letters that are all considered equivalent. Let $\#_k^{\hat{v}}$ denote the quantity $\sum_{v \in \hat{v}} (\#_k^v)$, that is the total number of occurrences of the values of $\hat{v}$ in column k. The bounds (11), (12), (14), (15) are generalised as follows:

$$ls_k^+(\hat{v}) = \max(0, \#_k^{\hat{v}} - \#_{k-1}^{\hat{v}}) \tag{17}$$

$$us_k^+(\hat{v}) = \#_k^{\hat{v}} - \max(0, \#_{k-1}^{\hat{v}} + \#_k^{\hat{v}} - R) \tag{18}$$

$$ls_k^-(\hat{v}) = \max(0, \#_k^{\hat{v}} - \#_{k+1}^{\hat{v}}) \tag{19}$$

$$us_k^-(\hat{v}) = \#_k^{\hat{v}} - \max(0, \#_{k+1}^{\hat{v}} + \#_k^{\hat{v}} - R) \tag{20}$$

and we get the following necessary conditions:

$$\sum_{k=0}^{K-1} s_k^+(\hat{v}) = \sum_{v \in \hat{v}} \sum_{r=0}^{R-1} S_r(v) \tag{21}$$

$$\sum_{k=0}^{K-1} s_k^-(\hat{v}) = \sum_{v \in \hat{v}} \sum_{r=0}^{R-1} S_r(v) \tag{22}$$

Note that (21) and (22) specialise respectively to (13) and (16) when $\hat{v} = \{v\}$.

3.3 Constraining the Minimum and Maximum Length of a Stretch

Suppose now that we have lower and upper bounds on the length of a stretch of a given value v for each row:

- $LLS(v)$ is the minimum length of a stretch of value v in every row.
- $ULS(v)$ is the maximum length of a stretch of value v in every row.

3.3.1 Necessary Conditions

We get the following necessary conditions:

$$\forall k \in [0, K-1] : \#_k^v \geq \sum_{j=\max(0, k-LLS(v)+1)}^{k} ls_j^+(v) \tag{23}$$

$$\forall k \in [0, K-1] : \#_k^v \geq \sum_{j=k}^{\min(K-1, k+LLS(v)-1)} ls_j^-(v) \tag{24}$$

The intuition behind (23) (respectively (24)) is that the stretches starting (respectively ending) at the considered columns j must overlap column k.

$$\begin{array}{l} \forall k \in [0, K-1-ULS(v)] : \\ ls_k^+(v) + \sum_{j=LLS(v)}^{ULS(v)} \#_{k+j}^v \leq (ULS(v) - LLS(v) + 1) \cdot R \end{array} \tag{25}$$

$$\begin{array}{l} \forall k \in [ULS(v), K-1] : \\ ls_k^-(v) + \sum_{j=LLS(v)}^{ULS(v)} \#_{k-j}^v \leq (ULS(v) - LLS(v) + 1) \cdot R \end{array} \tag{26}$$

The intuition behind (25) is as follows. For each stretch beginning at column k there must be an element distinct from v in a column $j \in [k + LLS(v), k + ULS(v)]$ of the same row. So the number of such values different from v in columns $[k + LLS(v), k + ULS(v)]$ (i.e., $ls_k^+(v)$) plus the number of occurrences of v in columns $[k + LLS(v), k + ULS(v)]$ (i.e., $\sum_{j=LLS(v)}^{ULS(v)} \#_{k+j}^v$) should not exceed the available space $(ULS(v) - LLS(v) + 1) \cdot R$. The reasoning for (26) is similar but considers stretches *ending* at column k.

Example 4 Figure 4 illustrates the necessary condition (25) on the minimum number of occurrences of values 0 and 1 in columns 2 and 3, provided that the minimum number of stretches of value 1 starting in column 0 is equal to 3 (i.e., $ls_0^+(1) = 3$), and that the minimum and maximum lengths of a stretch of value 1 are respectively equal to 2 and 3 (i.e., $LLS(1) = 2$ and $ULS(1) = 3$). In this context, inequality (25) holds since its left-hand side, i.e., the minimum number of occurrences of 0 and 1 in columns 2 and 3, is equal to $3 + (3 + 1)$, while its right-hand side, i.e., the available space in columns 2 and 3, is equal to $(3 - 2 + 1) \cdot 5$. □

3.3.2 Extension

We now provide another necessary condition, which holds for any value $v \in [0, V-1]$ and for any $ULS(v) + 1$ consecutive columns of the matrix $\mathcal{M}^{\#}$. Let $\Delta_{v,k,\ell}$ (with $v \in [0, V-1]$ and $k \in [0, K-\ell]$) denote the number of occurrences of values different from value v in any ℓ consecutive columns starting at column k of matrix $\mathcal{M}^{\#}$. Also, let $\Gamma_{u,k,\ell}$ (with $u \in [0, V-1]$ and $k \in [0, K-\ell]$) denote a lower bound on the minimum number of stretches of value u that for sure have at least $LLS(u)$ values within any ℓ consecutive columns starting at column k of matrix $\mathcal{M}$. Formally:

$$\Delta_{v,k,\ell} = R \cdot \ell - \sum_{i=k}^{k+\ell-1} \#_i^v \tag{27}$$

$$\Gamma_{u,k,\ell} = \max \{\#_i^u \mid k + LLS(u) - 1 \leq i \leq k + \ell - LLS(u)\} \tag{28}$$

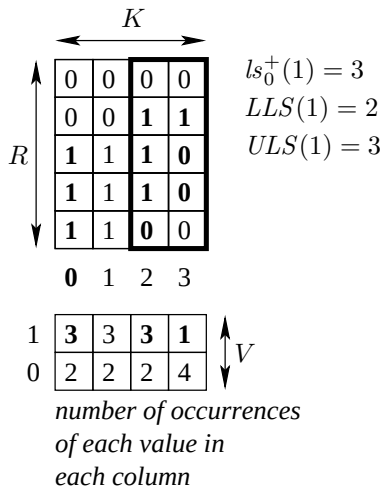

Figure 4 Minimum number of occurrences of values 0 and 1 in columns 2 and 3 with respect to (a) the minimum number of stretches starting in another column and (b) the minimum and maximum stretch lengths. In the $R \times K$ matrix, boldface in column 0 corresponds to the requirement $ls_0^+(1) = 3$, whereas boldface in the box corresponds to the left hand side of (25): boldface 0s correspond to the term $ls_0^+(1)$; boldface 1s correspond to the term $\sum_{j=2}^{3} \#_j^1$. Boldface in the $V \times K$ matrix corresponds to the occurrence constraints on value $v = 1$.

We get the following necessary condition:

$$\begin{array}{l} \forall v \in [0, V-1] : \forall k \in [0, K - ULS(v) - 1] : \\ R - \sum\limits_{\substack{u \in [0, V-1] \\ u \neq v}} \Gamma_{u,k,ULS(v)+1} \\ \leq \Delta_{v,k,ULS(v)+1} - \sum\limits_{\substack{u \in [0, V-1] \\ u \neq v}} LLS(u) \cdot \Gamma_{u,k,ULS(v)+1} \end{array} \tag{29}$$

The left-hand side of (29) corresponds to the number of rows of matrix $\mathcal{M}$ that do not necessarily contain a stretch of length $LLS(u)$ for a value u different from v. The right-hand side of (29) corresponds to the number of occurrences of values different from value v that are not necessarily part of a stretch of length $LLS(u)$. If (29) does not hold, then we have a contradiction since at least one row of the matrix $\mathcal{M}$ contains more than $ULS(v)$ occurrences of value v. Figure 5(A) illustrates condition (29).

Example 5 Let us illustrate constraint (29) on an $R = 3$ by $K = 6$ matrix $\mathcal{M}$ of variables taking their values in the set $\{0, 1, 2, 3\}$ (i.e., $V = 4$). For this purpose, assume that the numbers of occurrences of 0, 1, 2, 3 in the six consecutive columns of $\mathcal{M}$, as well as the minimum and maximum stretch lengths of values 0, 1, 2, 3 are respectively equal to:

- $\#_{0..5}^3 = [1, 0, 1, 2, 1, 2]$, $LLS(3) = 1$, $ULS(3) = 2$
- $\#_{0..5}^2 = [0, 0, 0, 0, 0, 0]$, $LLS(2) = 3$, $ULS(2) = 3$

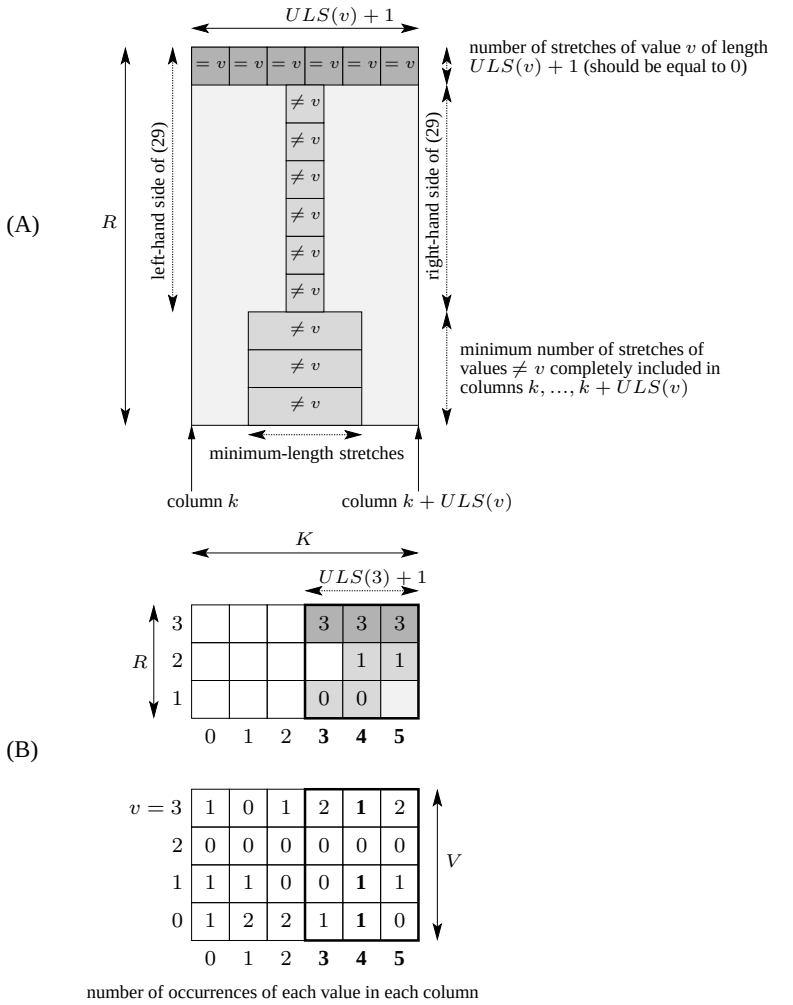

Figure 5 (A): Illustration of necessary condition (29). (B): Illustration of Example 5, where a too long stretch of value 3 occurs in columns 3 to 5 since, in these columns, the two occurrences of 0 (respectively 1) have to form a stretch; numbers in boldface respectively denote the columns we focus on (the last three columns) and the number of occurrences of values we focus on (the number of occurrences of values 0, 1, and 3 in column 4).

- $\#^1_{0..5} = [1,1,0,0,1,1], \quad LLS(1) = 2, \quad ULS(1) = 2$
- $\#^0_{0..5} = [1,2,2,1,1,0], \quad LLS(0) = 2, \quad ULS(0) = 4$

See Figure 5(B): we focus on value $v = 3$ and on the collection $\mathcal{C}$ of $ULS(3)+1 = 3$ consecutive columns of matrix $\mathcal{M}^{\#}$ that start at column 3 (recall that columns are numbered from 0). The number of occurrences of values different from value $v = 3$ within $\mathcal{C}$ is equal to $\Delta_{3,3,3} = 3 \cdot 3 - \sum_{i=3}^{3+3-1} \#^3_i = 9 - (2+1+2) = 4$. For each value u different from value $v = 3$ (i.e., values 0, 1, and 2), consider the minimum

number of stretches of value u that for sure have at least $LLS(u)$ values within $\mathcal{C}$. We have:

- $\Gamma_{0,3,3} = \max_{i=3+2-1}^{3+3-2} \#_i^u = \#_4^0 = 1$,
- $\Gamma_{1,3,3} = \max_{i=3+2-1}^{3+3-2} \#_i^u = \#_4^1 = 1$,
- $\Gamma_{2,3,3} = \max_{i=3+3-1}^{3+3-3} \#_i^u = 0$.

Finally, since the condition $3 - (1 + 1 + 0) = 1 \leq 0 = 4 - (2 \cdot 1 + 2 \cdot 1 + 3 \cdot 0)$ does not hold, the *matrix-of-automata-and-gcc* constraint pattern cannot be satisfied. This can be interpreted as the fact that, in the last three columns of matrix $\mathcal{M}$, there must be at least one row containing three consecutive occurrences of 3. This contradicts the requirement $ULS(3) = 2$. □

3.4 Combining Two String Properties: Forbidden Prefixes or Suffixes and Number of Stretches

One can also combine several string properties and get stronger conditions. For example, assume that the row automaton $\mathcal{A}$ has the following properties with respect to two distinct values u and v (with $u, v \in [0, V-1]$):

- The maximum number of stretches of value u is equal to 1.
- The word u^+v is a forbidden prefix.

We then have the following necessary condition:

$$\forall i \in [1, K-2] : \max(0, \#_0^u + \#_i^u - R) + \#_{i+1}^v \leq R \tag{30}$$

The quantity $\max(0, \#_0^u + \#_i^u - R)$ represents the minimum number of rows where value u for sure occurs both in columns 0 and i. Since we know that we can have at most one stretch of value u in a row, this means that we have at least $\max(0, \#_0^u + \#_i^u - R)$ stretches of value u starting at column 0. Hence (30) enforces that none of these stretches be directly followed by a v.

Similarly, when vu^+ is a forbidden suffix, we have that:

$$\forall i \in [1, K-2] : \max(0, \#_{K-1}^u + \#_i^u - R) + \#_{i-1}^v \leq R \tag{31}$$

Example 6 Let us illustrate (30) on an $R = 3$ by $K = 6$ matrix $\mathcal{M}$ of variables taking their values in the set $\{0, 1, 2\}$ (i.e., $V = 3$). For this purpose, assume that any three consecutive stretches within a row of $\mathcal{M}$ must be over the values $\{1\,0\,2, 0\,1\,2, 0\,1\,0\}$, and that the numbers of occurrences of 0, 1, 2 in the six columns of $\mathcal{M}$ are respectively equal to:

- $\#_{0..5}^0 = [1, 1, 1, 2, 1, 0]$
- $\#_{0..5}^1 = [2, 2, 2, 1, 2, 0]$
- $\#_{0..5}^2 = [0, 0, 0, 0, 0, 3]$

Consider values $u = 1$ and $v = 2$. Note that each row of matrix $\mathcal{M}$ contains at most one stretch of value 1. Moreover, the word 1^+2 cannot be the prefix of any row of $\mathcal{M}$. Now, focus on the two occurrences of value 1 both in columns 0 and 4 of matrix

$\mathcal{M}$, as well on the number $\#_5^2 = 3$ of occurrences of value 2 in the last column. We have that $\max(0, \#_0^1 + \#_4^1 - 3) + \#_5^2 = \max(0, 2 + 2 - 3) + 3 = 4$ is greater than $R = 3$, which is a contradiction since the word 1 1 1 1 1 2 will necessarily be a row of matrix $\mathcal{M}$. □

3.5 Constraining Value Precedence

Suppose now that we require that if a value v occurs at any position k in a row, then another value u also occur at least ℓ times (with $\ell > 0$) before position k in that row. This can be directly translated into the following necessary condition

$$\#_0^v = 0 \quad \wedge \quad \forall k \in [1, K-1] : \sum_{i=0}^{k-1} \#_i^u \geq \ell \cdot \#_k^v \tag{32}$$

where $\ell \cdot \#_k^v$ represents a lower bound on the number of occurrences of value u in columns $0, 1, \ldots, k-1$, under the hypothesis that we have $\#_k^v$ occurrences of value v on column k.

Value precedence, with $\ell = 1$, was originally introduced in [16] to break symmetries in the context where all occurrences of a value can be exchanged with all occurrences of another value, e.g., in graph colouring problems the colours can be exchanged unless additional constraints prevent this. Value precedence can also be extracted from an automaton and Section 3.7 describes how to perform this task automatically.

3.6 Replacing the gcc Column Constraint by a Sum Constraint

Assume that we want to replace the gcc constraint on a given column k by the requirement that the sum S of the variables of column k be in a given interval $[\ell, u]$. By first introducing cardinality variables on the column of the matrix $\mathcal{M}$ for denoting the number of occurrences of each value, and second linking the newly introduced cardinality variables to the sum S with a channelling constraint, we can directly reuse all the results previously introduced. For this purpose, besides setting the minimum and maximum value of S to ℓ and u, we create a *channelling* constraint of the form

$$S = 0 \cdot \#_k^0 + 1 \cdot \#_k^1 + \cdots + (V-1) \cdot \#_k^{V-1} \tag{33}$$

We can set all the previous necessary conditions on the newly introduced cardinality variables $\#_k^0, \#_k^1, \ldots, \#_k^{V-1}$.

3.7 Extracting Occurrence, Word, and Stretch Constraints from an Automaton, or How to Annotate an Automaton with String Properties

Toward automatically inferring the constant bounds $LW_r(w)$, $LWP_r(w)$, $LWS_r(w)$, $LS_r(w)$, etc., of the previous sub-sections, we now describe how a given automaton

$\mathcal{A}$ can be automatically annotated with counter variables constrained to reflect properties of the strings that the automaton recognises. This is especially useful if $\mathcal{A}$ is a product automaton for several constraints. For this purpose, we use the *automaton* constraint introduced in [3], which (unlike the *regular* constraint [19]) allows us to associate counters to a transition. Each string property requires (i) a counter variable whose final value reflects the value of that string property, (ii) possibly some auxiliary counter variables, (iii) initial values of the counter variables, and (iv) update formulae in the automaton transitions for the counter variables. We now give the details for some string properties.

In this context, n denotes an integer or a decision variable, b denotes a 0/1 integer or decision variable, $\hat{v}$ denotes a set of letters, $\hat{v}^+$ denotes a nonempty sequence of letters in $\hat{v}$, and s_i denotes the letter at position i of word s. We describe the annotation for the following string properties for any given string:

- $wordocc(\hat{v}^+, n)$: Word $\hat{v}^+$ occurs n times.
- $wordprefix(\hat{v}^+, b)$: $b = 1$ if and only if word $\hat{v}^+$ is a prefix of the string.
- $wordsuffix(\hat{v}^+, b)$: $b = 1$ if and only if word $\hat{v}^+$ is a suffix of the string.
- $stretchocc(\hat{v}, n)$: Stretches of letters in set $\hat{v}$ occur n times.
- $stretchminlen(\hat{v}, n)$: If letters in set $\hat{v}$ occur, then n is the length of the shortest such stretch, otherwise $n = +\infty$.
- $stretchmaxlen(\hat{v}, n)$: If letters in set $\hat{v}$ occur, then n is the length of the longest such stretch, otherwise $n = 0$.
- $valueprec(x, y, n)$: If y occurs, then x occurs n times before the first occurrence of y, otherwise $n = 0$.

For a given annotation, Table 1 shows which counters it introduces, their initial and final values, as well as the formulae for counter updates to be used in the transitions. Figure 1 shows an automaton annotated for $stretchocc(\{0\}, n)$.

An automaton can be annotated with multiple string properties—since annotations do not interfere with one another—and can be simplified in order to remove multiple occurrences of identical counters that come from different string properties.

It is worth noting that propagation is possible from the decision variables to the counter variables, and vice-versa.

3.8 Heuristics for Selecting Relevant String Properties for an Automaton

In our experiments (see Section 6), we chose to look for the following string properties:

- For each letter, lower and upper bounds on the number of its occurrences.
- For each letter, lower and upper bounds on the number and length of its stretches.
- Each word of length at most 3 that cannot occur at all.
- Each word of length at most 3 that cannot occur as a prefix or suffix.

These properties are derived, one at a time, as follows. We annotate the automaton as described in the previous sub-section by the candidate string property. Then we compute by labelling the feasible values of the counter variable reflecting the given property, giving up if the computation does not finish within 5 CPU seconds. Among the

collected word, prefix, suffix, and stretch properties, some properties are subsumed by others and are thus filtered away. Other properties could certainly have been derived, e.g., not only forbidden words, but also bounds on the number of occurrences of words. Our choice was based on two considerations: first which properties we are able to derive necessary conditions for, and second empirical observations of what actually pays off in our benchmarks.

4 The Cardinality Automaton of an Automaton

The previous section introduced different complementary ways of generating necessary conditions (expressed in terms of arithmetic constraints) from a given automaton for the row constraints of the matrix $\mathcal{M}$ when its columns are subject to gcc or *sum* constraints. This section presents an orthogonal systematic approach, again based on double counting, which can handle the same class of column constraints completely mechanically, without first having to choose relevant string properties.

Consider an $R \times K$ matrix $\mathcal{M}$, where in each row we have the same constraint, represented by an automaton $\mathcal{A}$ of p states $s_0, \ldots, s_{p-1}$, and in each column we have a gcc or linear (in)equality constraint where all the coefficients are the same. We will first construct an automaton that simulates the parallel running of the R copies of $\mathcal{A}$ and consumes entire columns of $\mathcal{M}$ at each transition. Since this new automaton has p^R states, we then use an abstraction where we just *count* the number of automata that are in each state of $\mathcal{A}$. As even this abstracted automaton has a size exponential in p, we then use a linear-size encoding with linear constraints that allows us to consider the column constraints on $\mathcal{M}$ as well.

4.1 Necessary Row Constraints

The *vector automaton* $\overline{\mathcal{A}_R}$ consumes column vectors of size R at each transition. Its states are sequences of R states of $\mathcal{A}$, where sequence entry ℓ is the state of the automaton of row ℓ. There is a transition from state $\langle s_{i_0}, \ldots, s_{i_{R-1}} \rangle$ to state $\langle s_{j_0}, \ldots, s_{j_{R-1}} \rangle$ if and only if for each ℓ there is a transition in $\mathcal{A}$ from s_{i_ℓ} to s_{j_ℓ}. A state $\langle s_{i_0}, \ldots, s_{i_{R-1}} \rangle$ is initial (respectively accepting) if each of the s_{i_ℓ} is the initial (respectively an accepting) state of $\mathcal{A}$.

For example, in Figure 6 (top) is the vector automaton $\overline{\mathcal{C}_2}$ for the (counter-free version of the) automaton $\mathcal{C}$ (with $p = 3$ states) in Figure 1 for the $global\_contiguity$ constraint and vectors of $R = 2$ elements over the set $\{0, \ldots, V-1\}$ for $V = 2$ values. Each state is an R-tuple of states of $\mathcal{C}$, indicating in which states of $\mathcal{C}$ the R copies of $\mathcal{C}$ respectively are. There are $p^R = 3^2 = 9$ states, each with at most $V^R = 2^2 = 4$ outgoing transitions, hence the size of the cardinality automaton is exponential in the number p of states of the original automaton. So let us just *count* the number of copies of the original automaton that are in each of its states: this leads to the following concept.

The *cardinality* (*vector*) *automaton* $\#\left(\overline{\mathcal{A}_R}\right)$ is an abstraction of the vector automaton $\overline{\mathcal{A}_R}$ that also consumes column vectors of size R at each transition. Its states

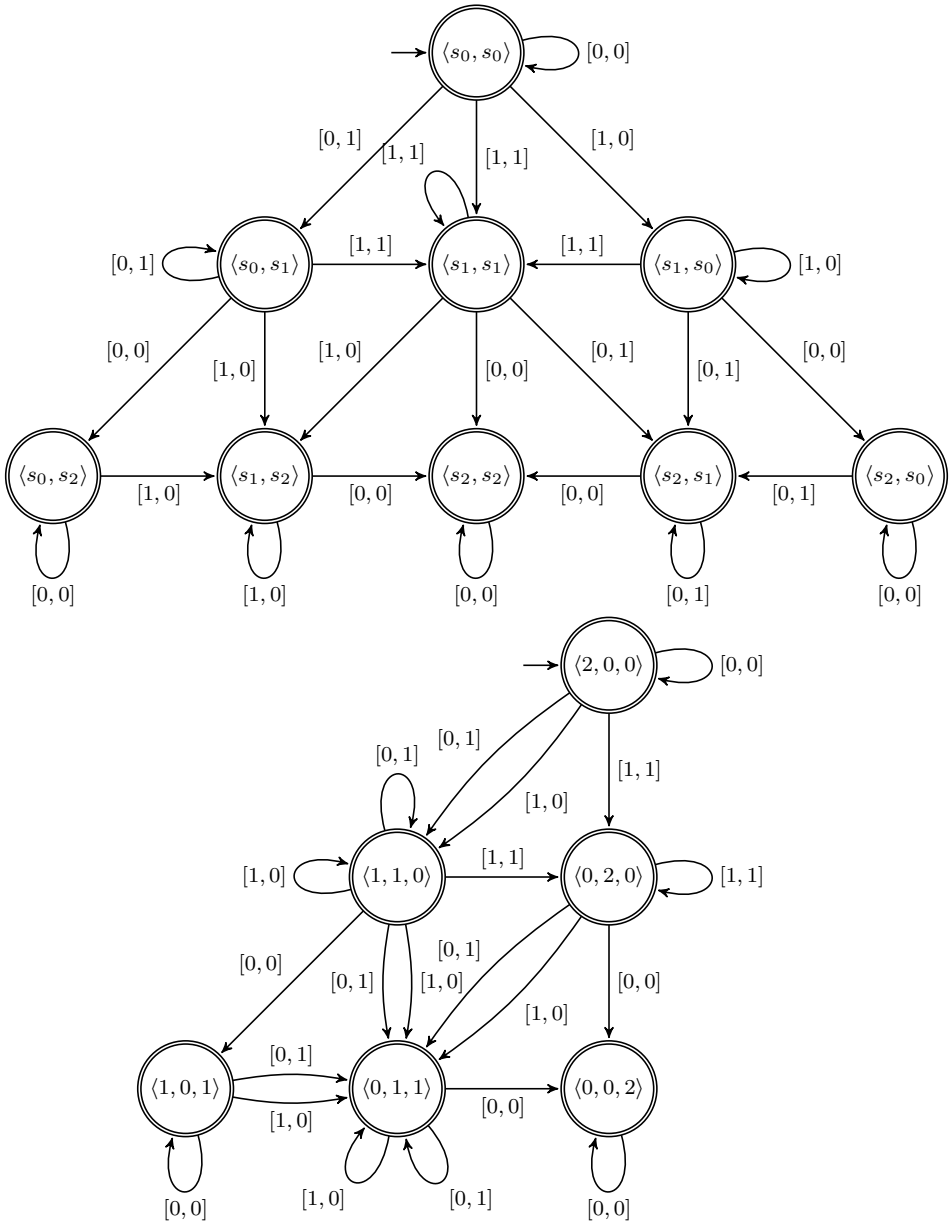

Figure 6 (Top): Vector automaton $\overline{\mathcal{C}_2}$ for the (counter-free version of the) automaton $\mathcal{C}$ (with $p = 3$ states) in Figure 1 for the $global\_contiguity$ constraint and vectors of $R = 2$ elements over the set $\{0, 1\}$. (Bottom): Cardinality automaton $\#\left(\overline{\mathcal{C}_2}\right)$ for the automaton $\mathcal{C}$ and vectors of $R = 2$ elements.

are sequences of p numbers, whose sum is R, where entry i is the number of automata $\mathcal{A}$ in state s_i. There is a transition from state $\langle c_{i_0}, \ldots, c_{i_{p-1}} \rangle$ to state $\langle c_{j_0}, \ldots, c_{j_{p-1}} \rangle$ if and only if there exists a multiset of R transitions in $\mathcal{A}$ such that for each ℓ there are c_{i_ℓ} of these R transitions going out from s_ℓ, and for each m there are c_{j_m} of these R transitions arriving into s_m. A state $\langle c_{i_0}, \ldots, c_{i_{p-1}} \rangle$ is initial (respectively accepting) if $c_{i_\ell} = 0$ whenever s_ℓ is not the initial (respectively an accepting) state of $\mathcal{A}$.

For example, in Figure 6 (bottom) is the cardinality automaton $\#\left(\overline{\mathcal{C}_2}\right)$ for the automaton $\mathcal{C}$ (with $p = 3$ states) and vectors of $R = 2$ elements. Each state is a p-tuple of natural numbers, indicating how many of the R copies of $\mathcal{C}$ are in each state of $\mathcal{C}$. For instance, states $\langle s_0, s_1 \rangle$ and $\langle s_1, s_0 \rangle$ of $\overline{\mathcal{C}_2}$ are merged into state $\langle 1, 1, 0 \rangle$. Note that this cardinality automaton is non-deterministic. In general, the number of states of $\#\left(\overline{\mathcal{A}_R}\right)$ is the number of ordered partitions of p, and thus exponential in p.

However, it is possible to have a compact encoding of $\#\left(\overline{\mathcal{A}_R}\right)$ via constraints. Toward this, we use $p \cdot (K+1)$ decision variables S_i^k in the domain $\{0, 1, \ldots, R\}$ to encode the states of an arbitrary path of length K (the number of columns) in $\#\left(\overline{\mathcal{A}_R}\right)$. We call S_i^k a *state-count variable*: it denotes the number of automata $\mathcal{A}$ that are in state s_i after column $k-1$ has been consumed; for $k \in \{1, \ldots, K\}$, the sequence $\langle S_0^k, S_1^k, \ldots, S_{p-1}^k \rangle$ has as possible values the states of $\#\left(\overline{\mathcal{A}_R}\right)$ after the latter has consumed column $k-1$ in one transition; for $k = 0$, the sequence $\langle S_0^0, S_1^0, \ldots, S_{p-1}^0 \rangle$ is fixed to $\langle R, 0, \ldots, 0 \rangle$ when, without loss of generality, s_0 is the initial state of $\mathcal{A}$. We get the following constraint for column k:

$$S_0^k + S_1^k + \cdots + S_{p-1}^k = R \tag{34}$$

and the following additional constraint for the last column K:

$$\forall i \in \{0, \ldots, p-1\} : S_i^K = 0 \leftarrow s_i \text{ is not an accepting state of } \mathcal{A} \tag{35}$$

Assume that $\mathcal{A}$ has a set $\mathcal{T} = \{(a_0, \ell_0, b_0), (a_1, \ell_1, b_1), \ldots, (a_{q-1}, \ell_{q-1}, b_{q-1})\}$ of q transitions, where transition (a_i, ℓ_i, b_i) goes from state $a_i \in \{s_0, s_1, \ldots, s_{p-1}\}$ to state $b_i \in \{s_0, s_1, \ldots, s_{p-1}\}$ upon reading letter $\ell_i \in \{0, 1, \ldots, V-1\}$. We use $q \cdot K$ decision variables T_i^k in the domain $\{0, 1, \ldots, R\}$ to encode the transitions of an arbitrary path of length K in $\#\left(\overline{\mathcal{A}_R}\right)$. We call T_i^k a *transition-count variable*: it denotes the number of automata $\mathcal{A}$ that trigger the transition t_i after column k has been consumed, with $k \in \{0, \ldots, K-1\}$. We get the following constraint for column k:

$$T^k_{(a_0,\ell_0,b_0)} + T^k_{(a_1,\ell_1,b_1)} + \cdots + T^k_{(a_{q-1},\ell_{q-1},b_{q-1})} = R \tag{36}$$

Consider two state encodings $\langle S_0^k, S_1^k, \ldots, S_{p-1}^k \rangle$ and $\langle S_0^{k+1}, S_1^{k+1}, \ldots, S_{p-1}^{k+1} \rangle$, and consider the transition encoding $\langle T^k_{(a_0,\ell_0,b_0)}, T^k_{(a_1,\ell_1,b_1)}, \ldots, T^k_{(a_{q-1},\ell_{q-1},b_{q-1})} \rangle$ between these two state encodings (with $0 \leq k < K$). To encode paths of length K in $\#\left(\overline{\mathcal{A}_R}\right)$, we introduce the following constraints. First, we constrain the number of automata $\mathcal{A}$ at any state s_j before reading column k to be equal to the number of firing transitions going out from s_j when reading column k:

$$\forall j \in \{0, \ldots, p-1\} : S_j^k = \sum_{(a_i,\ell_i,b_i) \in \mathcal{T} \,:\, a_i = s_j} T^k_{(a_i,\ell_i,b_i)} \tag{37}$$

Second, we constrain the number of automata $\mathcal{A}$ at state s_j after reading column k to be equal to the number of firing transitions coming into s_j when reading column k:

$$\forall j \in \{0, \ldots, p-1\} : S_j^{k+1} = \sum_{(a_i, \ell_i, b_i) \in \mathcal{T} \,:\, b_i = s_j} T^k_{(a_i, \ell_i, b_i)} \tag{38}$$

These constraints will be illustrated in an example in the next sub-section. A reformulation with linear constraints when $R = 1$ and there are *no* column constraints is described in [9].

4.2 Necessary Column Constraints and Channelling Constraints

The necessary constraints above on the state-count and transition-count variables only handle the row constraints, but they can also be used to handle column constraints of the previously considered kinds. These necessary constraints can thus be seen as a communication channel for enhancing the propagation between row and column constraints.

If column k has a gcc, then we constrain the number of occurrences of value v in column k to be equal to the number of transitions on v when reading column k:

$$\forall v \in \{0, \ldots, V-1\} : \#^v_k = \sum_{(a_i, \ell_i, b_i) \in \mathcal{T} \,:\, \ell_i = v} T^k_{(a_i, \ell_i, b_i)} \tag{39}$$

If column k has a constraint on its sum, then we constrain that sum to be equal to the value-weighted number of transitions on value v when reading column k:

$$\sum_{r=0}^{R-1} \mathcal{M}[r,k] = \sum_{v=0}^{V-1} v \cdot \left(\sum_{(a_i, \ell_i, b_i) \in \mathcal{T} \,:\, \ell_i = v} T^k_{(a_i, \ell_i, b_i)} \right) \tag{40}$$

Example 7 Consider an $R \times K$ matrix $\mathcal{M}$ with a $global\_contiguity$ constraint on each row and a gcc constraint on each column (see Example 1). An automaton $\mathcal{C}$ associated with the $global\_contiguity$ constraint is described by Figure 1. It has $p = 3$ states s_0, s_1, s_2 and $q = 5$ transitions $t_0 = (s_0, 0, s_0)$, $t_1 = (s_0, 1, s_1)$, $t_2 = (s_1, 1, s_1)$, $t_3 = (s_1, 0, s_2)$, $t_4 = (s_2, 0, s_2)$ labelled by values 0 and 1.

The encoding of $\#\left(\overline{\mathcal{C}_R}\right)$ has $p \cdot (K+1)$ state-count variables S_i^k such that constraint (34) is imposed: $S_0^k + S_1^k + S_2^k = R$ for every k. Since s_0 is the initial state of $\mathcal{C}$, we require that $S_0^0 = R$ and $S_1^0 = 0 = S_2^0$. Since $\mathcal{C}$ only has accepting states, no S_j^K is required to be zero under constraint (35). The encoding also has $q \cdot K$ transition-count variables T_i^k such that constraint (36) is imposed: $T_0^k + T_1^k + T_2^k + T_3^k + T_4^k = R$ for every k.

For instance, for $R = 3$ and $K = 7$, if $\mathcal{M}$ is

	0	1	2	3	4	5	6
0	0	0	0	1	1	1	1
1	0	1	1	1	0	0	0
2	1	1	0	0	0	0	0

then the state-count variable matrix S and transition-count variable matrix T respectively are

	0	1	2	3	4	5	6	7
s_0: 0	3	2	1	1	0	0	0	0
s_1: 1	0	1	2	1	2	1	1	1
s_2: 2	0	0	0	1	1	2	2	2

	0	1	2	3	4	5	6
t_0: 0	2	1	1	0	0	0	0
t_1: 1	1	1	0	1	0	0	0
t_2: 2	0	1	1	1	1	1	1
t_3: 3	0	0	1	0	1	0	0
t_4: 4	0	0	0	1	1	2	2

and they satisfy the constraints (34) to (36). The following three sets of linear constraints link the S and T variable matrices for every column k (with $0 \leq k < K$) and respectively are the necessary constraints (37), (38), and (39):

$$
\begin{array}{ll}
S_0^k = T_0^k + T_1^k & \text{(transitions that exit state } s_0\text{)} \\
S_1^k = T_2^k + T_3^k & \text{(transitions that exit state } s_1\text{)} \\
S_2^k = T_4^k & \text{(transitions that exit state } s_2\text{)} \\
\\
S_0^{k+1} = T_0^k & \text{(transitions that enter state } s_0\text{)} \\
S_1^{k+1} = T_1^k + T_2^k & \text{(transitions that enter state } s_1\text{)} \\
S_2^{k+1} = T_3^k + T_4^k & \text{(transitions that enter state } s_2\text{)} \\
\\
\#_k^0 = T_0^k + T_3^k + T_4^k & \text{(transitions labelled by value } 0\text{)} \\
\#_k^1 = T_1^k + T_2^k & \text{(transitions labelled by value } 1\text{)}
\end{array}
$$

Assume the gcc constraints on the columns of matrix $\mathcal{M}$ are as follows:

- Columns 0, 2, 4, 5, and 6 of $\mathcal{M}$ must each contain two 0s and a single 1.
- Columns 1 and 3 of $\mathcal{M}$ must each contain two 1s and a single 0.

The previously given instance of $\mathcal{M}$ satisfies these gcc constraints. Setting the cardinality variables $\#_k^v$ (with $0 \leq k < 7$ and $0 \leq v \leq 1$) according to these gcc constraints, the eight necessary constraints above are satisfied. Note that the necessary constraint (40) is not applicable here, as it is used when the column constraint is a summation constraint.

Now revise the gcc constraint on column 5 so that the latter is required to contain two 1s and a single 0, instead of vice-versa: we get the gcc constraints of Example 1:

- Columns 0, 2, 4, and 6 of $\mathcal{M}$ must each contain two 0s and a single 1.
- Columns 1, 3, and 5 of $\mathcal{M}$ must each contain two 1s and a single 0.

Revising the cardinality variables $\#_5^v$ (with $0 \leq v \leq 1$) accordingly, the system of linear constraints (34) to (39) fails when we post it using standard propagation on each constraint independently. □

For even more propagation, we can link the state-count variables S_i^k and transition-count variables T_i^k to the *state variables* and *transition variables* that are induced by the decomposition of the R automata $\mathcal{A}$, as discussed in [3]. For this purpose, let the state variables $Q_i^0, Q_i^1, \ldots, Q_i^K$ (with $0 \leq i < R$) denote the $K+1$ states visited by the automaton $\mathcal{A}$ on row i of length K. We get the following necessary gcc constraint

on column k (with $k \in \{0, \ldots, K\}$) of this matrix Q of state variables and the matrix S of state-count variables:

$$gcc\left(\left\langle Q_0^k, Q_1^k, \ldots, Q_{R-1}^k\right\rangle, \left\langle 0 : S_0^k, 1 : S_1^k, \ldots, p-1 : S_{p-1}^k\right\rangle\right) \tag{41}$$

Similarly, let the transition variables $E_i^0, E_i^1, \ldots, E_i^{K-1}$ denote the K triggered transitions of the automaton $\mathcal{A}$ on row i of length K. We get the following necessary gcc constraint on column k (with $k \in \{0, \ldots, K-1\}$) of this matrix E of transition variables and the matrix T of transition-count variables:

$$gcc\left(\left\langle E_0^k, E_1^k, \ldots, E_{R-1}^k\right\rangle, \left\langle 0 : T_0^k, 1 : T_1^k, \ldots, q-1 : T_{q-1}^k\right\rangle\right) \tag{42}$$

4.3 Incomparability of Filtering by Cardinality Automaton and String Properties

The filtering by the cardinality automaton and the filtering by the string properties are incomparable, as shown in the following example.

Example 8 Take a 3×6 matrix $\mathcal{M}$ of 0/1 variables (i.e., $R = 3$, $K = 6$, $V = 2$), where each row must be a word of the form $0^+1^+0^+1^+$ or $1^+0^+1^+0^+$ (i.e., we have two stretches of zeros and two stretches of ones). Assume that the numbers of occurrences of 0 and 1 in the six columns of $\mathcal{M}$ are respectively $\#_{0..5}^0 = [1, 0, 1, 1, 2, 1]$ and $\#_{0..5}^1 = [2, 3, 2, 2, 1, 2]$. The filtering by the cardinality automaton finds a contradiction without labelling on the variables of $\mathcal{M}$, but the filtering by the string properties (i.e., two stretches of zeros and two stretches of ones) does not. The converse happens when $\#_{0..5}^0 = [1, 0, 2, 2, 0, 1]$ and $\#_{0..5}^1 = [2, 3, 1, 1, 3, 2]$.

5 A Chain of Lexicographic Ordering Constraints Combined with Automaton Constraints

Let us again consider an R by K matrix of variables $\mathcal{M}$ where on each row we have a constraint specified by an automaton. Contrary to the previous sections, the automata here need not be the same for all the rows. Moreover we require that the rows be lexicographically ordered from the first to the last row. This is a natural way to break symmetries in the context of rostering problems, where each row corresponds to the schedule of an employee. Without loss of generality, we assume a non-strict lexicographic ordering constraint. Special cases of this pattern were already considered in the lex_chain constraint of SICStus Prolog [7,8], where the additional constraints on the vectors were $increasing$ and $among$. In that context, no guarantees were given about achieving domain consistency.

The contributions of this section are theoretical. First, we allow any constraint that can be expressed by an automaton without counters. Second, we guarantee domain consistency for this pattern.

We first sketch the basic filtering algorithm of the lex_chain constraint presented in [7, Section 5] (see Section 5.1). Since this algorithm relies on feasible lower and upper bounds being required for each vector, we then show how to compute the least

vector that is both greater than or equal to a given fixed vector and accepted by a given automaton (in Section 5.2). Finally we show how to adapt the basic filtering algorithm in order to handle the extra automaton constraints on the vectors (see Section 5.3).

5.1 Basic Filtering Algorithm of the lex_chain Constraint

The basic filtering algorithm of the lex_chain constraint consists of three steps:

1. Scan the vectors from the first to the last one and compute for each vector a feasible lower bound with respect to the domains of the variables and the feasible lower bound of the previous vector, if any.
2. Scan the vectors from the last to the first one and compute for each vector a feasible upper bound with respect to the domains of the variables and the feasible upper bound of the next vector, if any.
3. Filter each vector according to the requirement that it be located between two fixed feasible bounds. This can be done by using the $between$ constraint [7], which enforces a sequence of variables to be lexicographically greater than or equal to a fixed lower bound and less than or equal to a fixed upper bound.

5.2 Computing the Least Vector with respect to a Fixed Lower Bound and an Automaton Constraint

In addition to the lexicographic ordering constraints between adjacent rows of the matrix $\mathcal{M}$, we have an automaton constraint on each row of $\mathcal{M}$. Consequently, we have to compute during the first and second steps of the filtering algorithm (recalled in Section 5.1) lower and upper bounds that are feasible also with respect to the automaton constraint on the considered row. Without loss of generality, we show how to compute a feasible *lower* bound with respect to an automaton constraint.

Given an automaton $\mathcal{A}$ and a vector $\mathcal{V}$ that must satisfy $\mathcal{A}$ and be lexicographically greater than or equal to a fixed bound $[\ell_0, \ell_1, \ldots, \ell_{K-1}]$, we show how to compute the least vector $[a_0, a_1, \ldots, a_{K-1}]$ that is greater than or equal to $[\ell_0, \ell_1, \ldots, \ell_{K-1}]$ and satisfies $\mathcal{A}$ such that for all k in $[0, K-1]$ we have that a_k is in the domain of $\mathcal{V}[k]$ (i.e., step 1). We state that $\mathcal{V}[0]$ is greater than or equal to ℓ_0 and compute the minimum value v_0 of $\mathcal{V}[0]$ with respect to $\mathcal{A}$:

- If this new minimum value v_0 is strictly greater than ℓ_0, then we fix $\mathcal{V}[0]$ to v_0 and compute the corresponding least solution to $\mathcal{A}$ by successively fixing $\mathcal{V}[1], \mathcal{V}[2], \ldots, \mathcal{V}[K-1]$ to their minimum value and by propagating $\mathcal{A}$ after fixing each variable.
- If this new minimum value v_0 is equal to ℓ_0, then we fix $\mathcal{V}[0]$ to v_0 and reiterate the same process on variables $\mathcal{V}[1], \mathcal{V}[2], \ldots, \mathcal{V}[K-1]$.

Step 2 is performed in a similar way.

5.3 Filtering Algorithm of the lex_chain Constraint Combined with Automaton Constraints

The following filtering algorithm, called *Lex_chain_automaton*, of the lex_chain constraint combined with automaton constraints on the vectors, also consists of three steps:

1. Scan the vectors from the first to the last one and compute for each vector a feasible lower bound with respect to (i) the domains of the variables, (ii) the automaton constraint on that vector, and (iii) the feasible lower bound of the previous vector, if any.
2. Scan the vectors from the last to the first one and compute for each vector a feasible upper bound with respect to (i) the domains of the variables, (ii) the automaton constraint on that vector, and (iii) the feasible upper bound of the next vector, if any.
3. Filter each vector according to the requirement that it be located between two fixed feasible bounds and accepted by the automaton constraint of the considered vector. This can be done by computing the minimised product of the automaton of the $between$ constraint [7] and the automaton of the considered vector, and by filtering each vector with respect to this new automaton.

We now show that this algorithm achieves domain consistency.

Theorem 1 *Algorithm* Lex_chain_automaton *maintains domain consistency.*

Proof We show that if we set the variable at position k (with $0 \leq k < K$) of vector $\mathcal{V}$ to any remaining value of its domain, then we can always extend this to a full assignment that satisfies all the lexicographic ordering and automaton constraints in three steps:

1. We show how to fix completely vector $\mathcal{V}$, assuming $\mathcal{V}[k]$ is set to one of its potential values v. We compute the minimised product of the automaton of the $between$ constraint and the automaton $\mathcal{A}$ of the constraint on vector $\mathcal{V}$. We then use this automaton for finding a solution $[s_0, s_1, \ldots, s_{K-1}]$ where $s_k = v$ satisfies $\mathcal{A}$ as well as the required lower and upper bounds on vector $\mathcal{V}$.
2. All vectors that precede vector $\mathcal{V}$ can be fixed to their respective lower bounds, computed by the first step of the filtering algorithm. By construction, these lower bounds are all lexicographically smaller than or equal to vector $[s_0, s_1, \ldots, s_{K-1}]$.
3. We can also fix all vectors that follow vector $\mathcal{V}$ to their respective upper bounds computed, by the second step of the filtering algorithm. These upper bounds are all lexicographically greater than or equal to vector $[s_0, s_1, \ldots, s_{K-1}]$. □

6 Experimental Evaluation

NSPLib [21] is a very large repository of (artificially generated) instances of the *nurse scheduling problem* (NSP), which is about constructing a duty roster for nursing staff. Let R be the number of nurses, K the number of days of the scheduling horizon, and

V the number of shifts. The objective is to construct an $R \times K$ matrix of values in the integer interval $[0, V-1]$, with value $V-1$ representing the off-duty "shift".

In the *instance files*, there are hard *coverage constraints* and soft preference constraints; we only use the former here: they give for each day d and shift s the lower bound on the number of nurses that must be assigned to shift s on day d, and can be modelled by a global cardinality constraint (gcc) on the columns. Note that the gcc constraints on any two columns are in general *not* the same. There are instance files for $R \times 7$ rosters with $R \in \{25, 50, 75, 100\}$, and for $R \times 28$ rosters with $R \in \{30, 60\}$. There are three complexity indicators on the coverage constraints, giving rise to 270 instances for each of the 27 configurations of these indicators for the $R \times 7$ rosters, as well as to 80 instances for each of the 12 configurations of these indicators for the $R \times 28$ rosters.

In the *case files*, there are four hard constraints on the rows. For each shift s, there are lower and upper bounds on the number of occurrences of s in any row (the daily assignment of some nurse): this can be modelled by gcc constraints on the rows. There are also lower and upper bounds on the cumulative number of occurrences of the working shifts $0, \ldots, V-2$ in any row: this can be modelled by gcc constraints on the off-duty value $V-1$ and always gives tighter occurrence bounds on value $V-1$ than the previous gcc constraints. For each shift s, there are also lower and upper bounds on the length of any stretch of value s in any row: this can be modelled by $stretch\_path$ constraints on the rows. Finally, there are lower and upper bounds on the length of any stretch of the working shifts $0, \ldots, V-2$ in any row: this can be modelled by generalised $stretch\_path\_partition$ constraints [4] on the rows. Note that the constraints on any two rows are the *same*. There are 8 case files for the $R \times 7$ rosters, and another 8 case files for the $R \times 28$ rosters. Instead of posting four constraints on every row, we automatically generated (see [4] for details) deterministic finite automata (DFA) for the row constraints of each case, using their minimised product DFA (obtained through standard DFA algorithms) to achieve domain consistency on the conjunction of row constraints [3]. (Since we use the *automaton* constraint [3] rather than the *regular* constraint [19], the unfolding of the product automaton for a given number K of days is not an issue here, nor is the minimisation of the unfolded automaton.) For each case, string properties were automatically selected off-line as described in Section 3.8, and cardinality automata were automatically constructed off-line as described in Section 4, by using constraints (39) and (41). We can use (41) but not (42) since the SICStus implementation of the *automaton* constraint [8] uses explicit Q_i^k state variables but has no E_i^k transition variables.

Under these choices, the NSPLib benchmark corresponds to the pattern studied in this paper. To reduce the risk of reporting improvements where another search procedure can achieve much of the same impact, we use a two-phase search that exploits the fact that there is a single domain-consistent constraint on each row and column:

– Phase 1 addresses the column (coverage) constraints only: it seeks to assign enough nurses to given shifts on given days to satisfy *all but one* coverage constraint.

– In Phase 2, one column constraint and all row constraints remain to be satisfied. But these constraints form a Berge-acyclic CSP [1], and so the remaining decision variables can be easily labelled without search.

We cannot use the symmetry breaking method described in Section 5, for it would break the Berge-acyclicity in Phase 2. Instead, we break symmetries during search in Phase 1 by maintaining an equivalence relation: two rows (nurses) are in the same equivalence class while they are assigned to the same shifts and days.

This search procedure is much more efficient than row-wise labelling under decreasing value ordering (value $V - 1$ always has the highest average number of occurrences per row) combined with decreasing lexicographic ordering of the rows.

The objective of our experiments is to measure the impact in runtime and backtracks when using either or both of our methods. In the experiments, we used SICStus Prolog 4.2 on a quad core 2.8 GHz Intel Core i7-860 machine with 8MB cache per core, running Ubuntu Linux (using only one processor core). For each instance, we searched for its first solution, using a timeout of 1 CPU minute. For each case and nurse count R, we used the *first* 10 instances for each configuration of the NSPLib coverage complexity indicators, that is instances 1–270 for the $R \times 7$ rosters (Cases 1–8) and 1–120 for the $R \times 28$ rosters (Cases 9–16).

Table 2 summarises the running of these 3120 instances using neither, either, and both of our methods. Each row marked 'sat' (for satisfiable) for a given case and nurse count R shows the performance of each variant, namely the number of instances solved without timing out, as well as the total runtime (in seconds) and the total number of backtracks on all instances where *none* of the four variants timed out. Please note: this means that these totals are *comparable*, but also that they do not reveal any performance gains on instances where some variant(s) timed out. Similarly for each row marked 'unsat' (for unsatisfiable). Numbers in boldface indicate best performance in a row. Instance-wise plots of the runtimes are given in Figures 7 to 10; since for many runtimes there are multiple instances, the plots are made to *appear* to contain as many points as instances by multiplying every runtime for every variant by a new random number in the interval $[1.0, 1.3]$: the purpose of the plots is only to compare the *approximate* locations of the median runtimes for all variants.

It turned out that Cases 1–6, 9–10, and 12–14 are very simple (in the absence of preference constraints), so that our methods only decrease backtracks on one of those 2220 instances, but increase runtime. It also turned out that Case 11 is very difficult (even in the absence of preference constraints), so that even our methods systematically time out, because the product automaton of all row constraints is very big; we could have overcome this obstacle by using the built-in gcc constraint and the product automaton of the other two row constraints, but we wanted to compare all the cases under the same scenario. Hence we do not report any results on Cases 1–6 and 9–14.

Phase 1 uses dynamic choices, so the shape of the search tree can be affected by whether or not the necessary conditions generated by our methods are included. In a few cases, their inclusion does not yield the fewest backtracks.

An analysis of Table 2 and Figures 7 to 10 reveals that our methods decide more instances without timing out, and that they often drastically reduce the runtime and

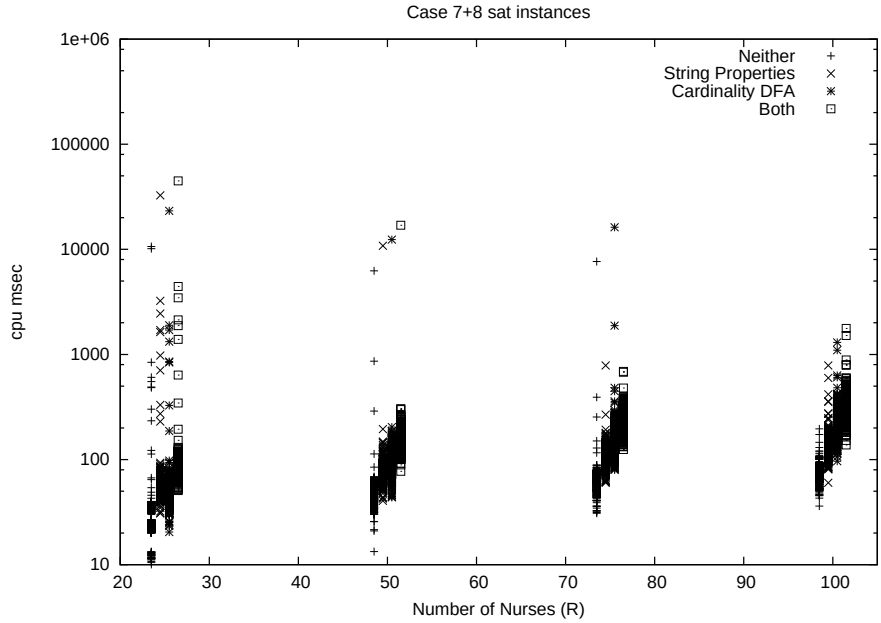

Figure 7 Runtimes (in milliseconds) of the *satisfiable* instances of NSPLib cases 7 and 8 using neither, either, or both of our methods.

number of backtracks (by up to four orders of magnitude), especially on the common unsatisfiable instances. However, runtimes are often increased (by up to one order of magnitude) on the common satisfiable instances. String properties are only rarely defeated by the cardinality DFA on any of the three performance measures, but their combination is often the overall winner, though rarely by a large margin. It would take a more fine-grained evaluation to understand when to use which string properties without increasing runtime on the satisfiable instances. The good performance of our methods on unsatisfiable instances is indicative of gains when exploring the whole search space, such as when solving an optimisation version of the problem or using soft (preference) constraints.

With constraint programming, NSPLib instances (without the soft preference constraints) were also used in [5, 6], but under row constraints different from those of the NSPLib case files that we used. NSP instances from a different repository were used in [18], though with soft global constraints: one of the insights reported there was the need for more interaction between the global constraints, and our paper shows steps that can be taken in that direction.

7 Conclusion

Since the necessary conditions generated by our methods are essentially linear constraints, these methods should be applicable also in the context of linear programming. Future work may also consider the integration of our techniques with the

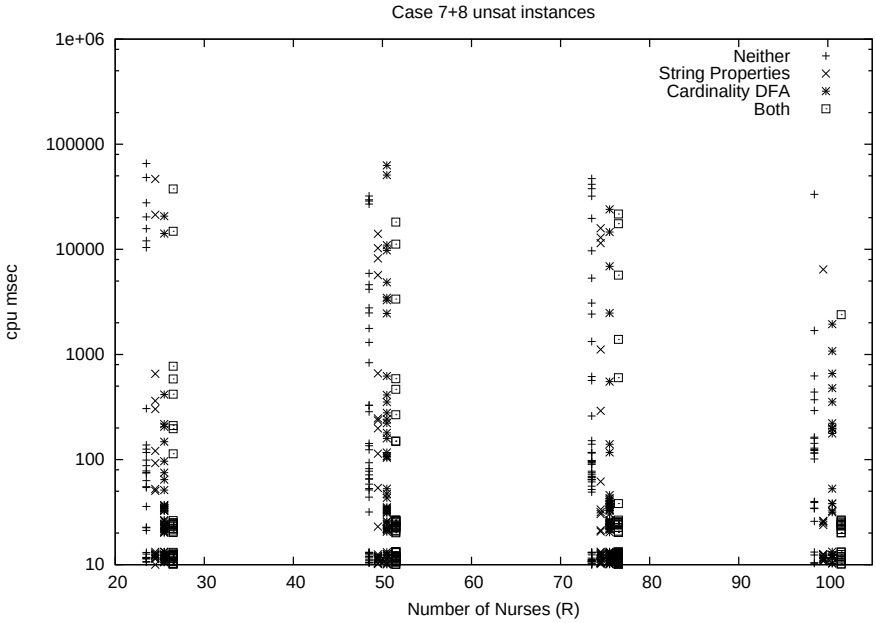

Figure 8 Runtimes (in milliseconds) of the *unsatisfiable* instances of NSPLib cases 7 and 8 using neither, either, or both of our methods.

multicost-regular constraint [17], which allows the direct handling of a gcc constraint in the presence of automaton constraints (as on the rows of NSPLib instances) without explicitly computing the product automaton, which can be very big.

Besides the fact that they can be used for generating necessary conditions for the *matrix-of-automata-and-gcc* pattern, annotated automata can be used for at least two other purposes:

- First, it is well known that making the product of several automata in order to achieve domain consistency for a conjunction of *automaton* constraints on the same sequence of variables usually leads to a size explosion. Now note that if we use the same set of string properties in order to annotate two automata that are applied to the same sequence of variables, then the variables corresponding to these string properties can act as a communication channel between these automata. By restricting the bounds of a given string property, an automaton communicates a partial view of its solution space to another automaton.
- Second, given a violated *matrix-of-automata-and-gcc* pattern, the necessary conditions generated from a given string property can capture a sharp explanation of the reason of failure. This kind of explanation is sharp for two reasons. On the one hand, by essence, the necessary conditions catch directly the interaction of the row and column constraints of the matrix. On the other hand, most necessary conditions typically point to a small subset of columns of the matrix as well as to specific cardinality variables of the gcc constraints. For instance, this is the case when the necessary condition corresponds to a forbidden word. This usually

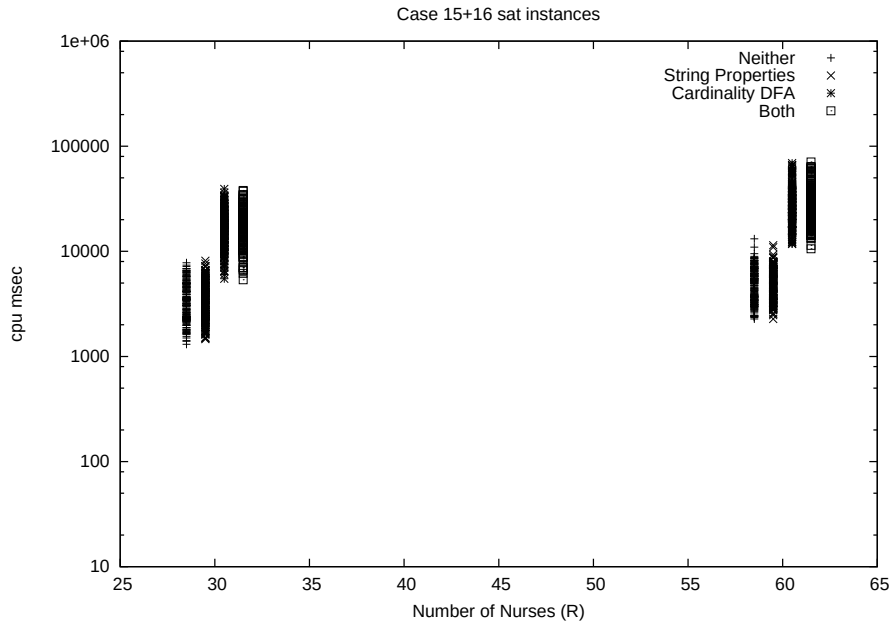

Figure 9 Runtimes (in milliseconds) of the *satisfiable* instances of NSPLib cases 15 and 16 using neither, either, or both of our methods.

provides a clear hint on how to relax the domains of the cardinality variables in order to achieve feasibility.

The tractability of propagating the *matrix-of-automaton-and-gcc* pattern of our [2] and the present extension thereof is studied in [14].

Acknowledgements We thank the referees, including the ones of CPAIOR'10, for their helpful comments, as well as Broos Maenhout for his replies to our questions on NSPlib. The last two authors were supported by grants 2007-6445 and 2011-6133 of the Swedish Research Council.

References

1. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. Journal of the ACM **30**, 479–513 (1983)
2. Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On matrices, automata, and double counting. In: A. Lodi, M. Milano, P. Toth (eds.) CPAIOR 2010, *LNCS*, vol. 6140, pp. 10–24. Springer (2010)
3. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: M.G. Wallace (ed.) CP 2004, *LNCS*, vol. 3258, pp. 107–122. Springer (2004)
4. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, 2nd Edition (revision a). Tech. Rep. T2012:03, Swedish Institute of Computer Science (2012). Available at `http://soda.swedish-ict.se/5195/`. The current working version of the catalogue is at `http://www.emn.fr/z-info/sdemasse/aux/doc/catalog.pdf`.
5. Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., Walsh, T.: SLIDE: A useful special case of the CARDPATH constraint. In: M. Ghallab, et al. (eds.) ECAI 2008, pp. 475–479. IOS Press (2008)
6. Brand, S., Narodytska, N., Quimper, C.G., Stuckey, P.J., Walsh, T.: Encodings of the *sequence* constraint. In: C. Bessière (ed.) CP 2007, *LNCS*, vol. 4741, pp. 210–224. Springer (2007)

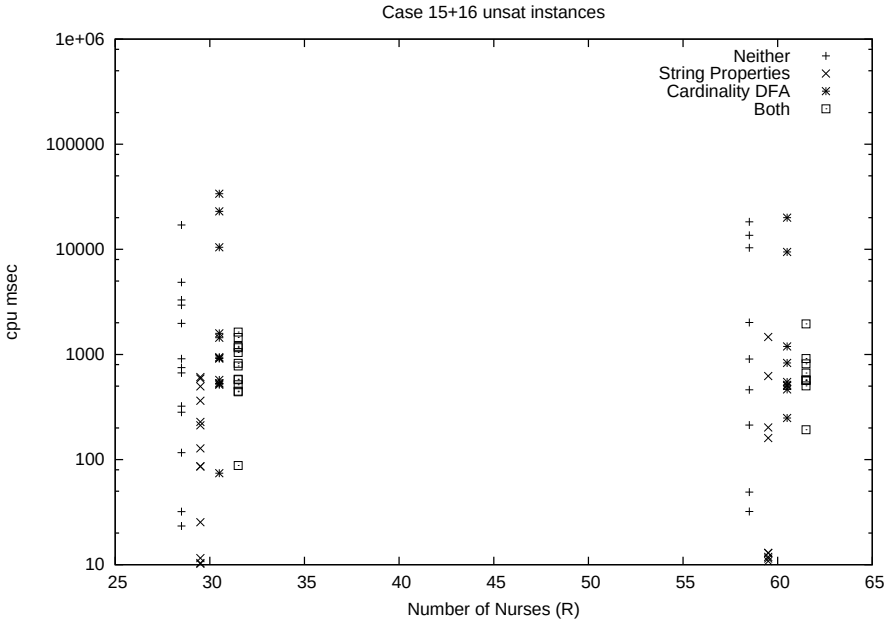

Figure 10 Runtimes (in milliseconds) of the *unsatisfiable* instances of NSPLib cases 15 and 16 using neither, either, or both of our methods.

7. Carlsson, M., Beldiceanu, N.: Arc-consistency for a *chain of lexicographic ordering* constraints. Tech. Rep. T2002-18, Swedish Institute of Computer Science (2002). URL `ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T--2002-18--SE.ps.Z`
8. Carlsson, M., et al.: SICStus Prolog User's Manual. Swedish Institute of Computer Science, 4.0 edn. (2007). URL `http://www.sics.se/sicstus/`
9. Côté, M.C., Gendron, B., Rousseau, L.M.: Modeling the *regular* constraint with integer programming. In: P. Van Hentenryck, L. Wolsey (eds.) CPAIOR 2007, *LNCS*, vol. 4150, pp. 29–43. Springer (2007)
10. Flener, P., Frisch, A.M., Hnich, B., Kızıltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: P. Van Hentenryck (ed.) CP 2002, *LNCS*, vol. 2470, pp. 462–476. Springer (2002)
11. Frisch, A.M., Hnich, B., Kızıltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In: P. Van Hentenryck (ed.) CP 2002, *LNCS*, vol. 2470, pp. 93–108. Springer (2002)
12. Frisch, A.M., Hnich, B., Kızıltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. Artificial Intelligence **170**(10), 803–834 (2006)
13. Frisch, A.M., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In: F. Rossi (ed.) CP 2003, *LNCS*, vol. 2833, pp. 318–332. Springer (2003)
14. de Haan, R., Narodytska, N., Walsh, T.: The RegularGcc matrix constraint. CoRR **abs/1201.0564** (2012)
15. Jukna, S.: Extremal Combinatorics. Springer (2001)
16. Law, Y.C., Lee, J.H.M.: Global constraints for integer and set value precedence. In: M.G. Wallace (ed.) CP 2004, *LNCS*, vol. 3258, pp. 362–376. Springer (2004)
17. Menana, J., Demassey, S.: Sequencing and counting with the *multicost-regular* constraint. In: W.J. van Hoeve, J.N. Hooker (eds.) CPAIOR 2009, *LNCS*, vol. 5547, pp. 178–192. Springer (2009)
18. Métivier, J.P., Boizumault, P., Loudni, S.: Solving nurse rostering problems using soft global constraints. In: I.P. Gent (ed.) CP 2009, *LNCS*, vol. 5732, pp. 73–87. Springer (2009)
19. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: M.G. Wallace (ed.) CP 2004, *LNCS*, vol. 3258, pp. 482–495. Springer (2004)
20. Régin, J.C., Gomes, C.: The *cardinality matrix* constraint. In: M.G. Wallace (ed.) CP 2004, *LNCS*, vol. 3258, pp. 572–587. Springer (2004)

21. Vanhoucke, M., Maenhout, B.: On the characterization and generation of nurse scheduling problem instances. European Journal of Operational Research **196**(2), 457–467 (2009). NSPLib is at `http://www.projectmanagement.ugent.be/nsp.php`

Annotation	Counter values	Counter updates
$wordocc(\hat{v}^+, n)$	$[0, ..., 0]$ $[c_1, ..., c_\ell]$ $[\_, ..., n]$	$[1, ...]$ if $u \in \hat{v}_1^+$ $[..., c_{i-1}, ...]$ if $1 < i < \ell \wedge u \in \hat{v}_i^+$ $[..., c_\ell + c_{\ell-1}]$ if $u \in \hat{v}_\ell^+$ $[..., 0, ...]$ if $0 < i < \ell \wedge u \notin \hat{v}_i^+$ $[..., c_\ell]$ if $u \notin \hat{v}_\ell^+$
	$c_i, i < \ell$ is 1 if and only if the most recently seen i letters match a prefix of $\hat{v}^+$. c_ℓ is the number of occurrences of words matching $\hat{v}^+$ so far.	
$wordprefix(\hat{v}^+, b)$	$[1, 0, ..., 0]$ $[c_0, c_1, ..., c_\ell]$ $[\_, ..., b]$	$[0, ..., c_{i-1}, ...]$ if $0 < i < \ell \wedge u \in \hat{v}_i^+$ $[0, ..., \max(c_\ell, c_{\ell-1})]$ if $u \in \hat{v}_\ell^+$ $[0, ..., 0, ...]$ if $0 < i < \ell \wedge u \notin \hat{v}_i^+$ $[0, ..., c_\ell]$ if $u \notin \hat{v}_\ell^+$
	c_0 is 1 if and only if the automaton is in the start state. $c_i, 0 < i < \ell$ is 1 if and only if the automaton has seen exactly i letters matching a prefix of $\hat{v}^+$. c_ℓ is 1 if and only if the first ℓ letters seen by the automaton match $\hat{v}^+$.	
$wordsuffix(\hat{v}^+, b)$	$[0, ..., 0]$ $[c_1, ..., c_\ell]$ $[\_, ..., b]$	$[1, ...]$ if $u \in \hat{v}_1^+$ $[..., c_{i-1}, ...]$ if $1 < i < \ell \wedge u \in \hat{v}_i^+$ $[..., c_{\ell-1}]$ if $u \in \hat{v}_\ell^+$ $[..., 0, ...]$ if $0 < i < \ell \wedge u \notin \hat{v}_i^+$ $[..., c_\ell]$ if $u \notin \hat{v}_\ell^+$
	c_i is 1 if and only if the most recently seen i letters match a prefix of $\hat{v}^+$.	
$stretchocc(\hat{v}, n)$	$[0, 0]$ $[c, d]$ $[n, \_]$	$[c - d + 1, 1]$ if $u \in \hat{v}$ $[c, 0]$ if $u \notin \hat{v}$
	c and d respectively denote the number of stretches of values matching $\hat{v}$ encountered so far, and whether or not the current position corresponds to values matching $\hat{v}$.	
$stretchminlen(\hat{v}, n)$	$[+\infty, +\infty, 0]$ $[c, d, e]$ $[n, \_, \_]$	$[\min(d, e + 1), d, e + 1]$ if $u \in \hat{v}$ $[c, c, 0]$ if $u \notin \hat{v}$
	c is the length of the shortest stretch of values matching $\hat{v}$ seen so far, or ∞ if no such stretch has been seen. d is the length of the shortest finished such stretch seen so far, or ∞ if no such stretch has been seen. e is the length so far of the current such stretch, or 0 otherwise.	
$stretchmaxlen(\hat{v}, n)$	$[0, 0]$ $[c, d]$ $[n, \_]$	$[\max(c, d + 1), d + 1]$ if $u \in \hat{v}$ $[c, 0]$ if $u \notin \hat{v}$
	c and d respectively denote the maximum length of the stretches of values matching $\hat{v}$ encountered so far, and the length of any such stretch corresponding to the current position.	
$valueprec(x, y, n)$	$[0, 0]$ $[c, d]$ $[n, \_]$	$[c, d + 1]$ if $x = u$ $[\max(c, d), -\infty]$ if $y = u$ $[c, d]$ if $x \neq u \neq y$
	c is 0 if no y has been seen, and the number of x's seen before the first y otherwise. d is the number of x's seen if no y has been seen, and $-\infty$ otherwise.	

Table 1 Given an annotation shown in the first column, the second column shows the counters used by the annotation: their initial values, their names, and their final values. The final value of one counter is the value computed by the annotation; the shared variable name indicates which one it is. Given a transition of the automaton reading letter u, the third column gives formulae for the counter updates performed in that transition, and under what conditions each given formula applies. For the first three annotations, ℓ is the word length. Finally, for each annotation, we give the interpretation of the respective counters.

				Neither			String Properties			Cardinality DFA			Both		
Case	R	Status	Found	#Inst	Time	#Bktk	#Inst	Time	#Bktk	#Inst	Time	#Bktk	#Inst	Time	#Bktk
7	25	sat	230	**230**	**30.1**	32109	**230**	47.4	13919	**230**	34.4	13823	**230**	66.1	**13791**
		unsat	38	37	94.5	113413	**38**	63.4	19491	**38**	**33.2**	21156	**38**	50.9	**12905**
7	50	sat	216	213	**16.1**	12165	**216**	24.6	**11055**	214	28.2	11077	**216**	44.3	11057
		unsat	43	40	88.6	79603	42	40.5	8678	**43**	104.3	60544	**43**	**32.8**	**5821**
7	75	sat	210	208	**18.6**	12709	209	20.8	628	**210**	41.9	12421	**210**	42.6	**340**
		unsat	48	**48**	103.7	155490	**48**	**35.8**	8858	**48**	42.0	12042	47	38.1	**8304**
7	100	sat	219	216	**13.0**	361	**219**	28.9	361	217	44.7	**355**	**219**	65.0	**355**
		unsat	26	22	37.1	8909	24	5.5	**452**	23	4.6	1000	**25**	**2.5**	459
8	25	sat	263	**263**	**6.3**	282	**263**	12.6	282	**263**	12.2	**76**	**263**	19.7	**76**
		unsat	7	**7**	96.2	121367	**7**	**0.1**	**19**	**7**	0.2	21	**7**	0.2	21
8	50	sat	259	**259**	**11.1**	**136**	**259**	16.8	**136**	**259**	24.0	**136**	**259**	36.3	**136**
		unsat	11	10	64.1	49358	**11**	4.8	715	10	52.0	29784	**11**	**3.4**	**592**
8	75	sat	246	245	**14.1**	449	245	23.1	**230**	**246**	39.2	449	**246**	53.6	**230**
		unsat	22	21	69.9	112880	**22**	**0.1**	**21**	**22**	0.5	62	**22**	0.3	30
8	100	sat	262	261	**17.4**	**239**	**262**	31.4	**239**	261	55.0	**239**	**262**	76.9	**239**
		unsat	6	4	0.3	73	**6**	**0.0**	**4**	4	0.4	73	**6**	0.1	**4**
15	30	sat	87	84	**171.2**	**37**	86	180.3	**37**	86	910.1	**37**	**87**	922.6	**37**
		unsat	23	9	23.5	2513	**23**	**1.5**	**9**	18	14.1	88	**23**	5.0	14
15	60	sat	87	**87**	**256.3**	**131**	**87**	271.4	**131**	**87**	1590.6	**131**	**87**	1616.1	**131**
		unsat	13	8	23.7	1001	**13**	**2.1**	**8**	11	31.4	394	**13**	5.2	12
16	30	sat	100	**100**	**391.8**	**153**	**100**	399.3	**153**	**100**	1907.0	**153**	**100**	1922.2	**153**
		unsat	10	5	7.8	172	**10**	**1.0**	**4**	6	51.4	167	**10**	4.3	6
16	60	sat	105	**105**	**578.5**	**145**	**105**	592.2	**145**	104	3217.7	**145**	**105**	3242.2	**145**
		unsat	3	1	16.9	579	**3**	**0.0**	**1**	2	0.7	2	**3**	0.7	2

Table 2 NSPlib benchmark results.